Parte SEIS

API de Data/Hora

Capítulo VINTE E UM

Classes Principais de Data/Hora

Objetivos do Exame

Criar e gerenciar eventos baseados em data e hora, incluindo uma combinação de data e hora em um único objeto usando LocalDate, LocalTime, LocalDateTime, Instant, Period e Duration.

Definir, criar e gerenciar eventos baseados em data e hora usando Instant, Period, Duration e TemporalUnit.

Uma nova API de Data/Hora

Desde o início do Java, java.util.Date e java.util.Calendar (introduzido posteriormente) têm sido as classes usadas ao se trabalhar com datas e horas.

No entanto, essas classes estão longe de serem perfeitas. Alguns de seus problemas são:

- java.util.Date representa o tempo com "apenas" precisão de milissegundos (o que pode não ser suficiente em algumas aplicações). Os anos começam em 1900 e os meses começam no 0.
- O fuso horário da data é o fuso horário padrão da JVM.
- Tanto java.util.Date quanto java.util.Calendar são classes mutáveis, o que significa que, quando elas mudam, não criam outra instância com os novos valores (o que não é ideal agora que você pode programar em um estilo funcional com Java).

Por essas razões, o Java 8 introduziu uma nova API de Data/Hora baseada na popular biblioteca Joda-Time e contida também no novo pacote java.time.

Este capítulo tratará das classes principais da nova API, que não fornecem informações sobre fuso horário. As classes que fornecem informações de fuso horário serão o tópico do próximo capítulo.

Vamos começar com uma visão geral de alto nível das classes principais.

Todas essas classes são imutáveis, seguras para threads e, com exceção de Instant, não armazenam ou representam um fuso horário.

Por um lado, temos:

LocalDate
Representa uma data com as informações de ano, mês e dia do mês. Por exemplo, 2015-08-25.

LocalTime
Representa uma hora com informações de hora, minutos, segundos e nanossegundos. Por exemplo, 13:21.05.123456789.

LocalDateTime
Uma combinação dos anteriores. Por exemplo, 2015-08-25 13:21.05.12345.

Por outro lado:

Instant
Representa um ponto único no tempo em segundos e nanossegundos. Por exemplo, 923.456.789 segundos e 186.054.812 nanossegundos.

Period
Representa uma quantidade de tempo em termos de anos, meses e dias. Por exemplo, 5 anos, 2 meses e 9 dias.

Duration
Representa uma quantidade de tempo em termos de segundos e nanossegundos. Por exemplo, 12,87656 segundos.

LocalDate, LocalTime, LocalDateTime e Instant implementam a interface java.time.temporal.Temporal, portanto todos têm métodos semelhantes.

Enquanto Period e Duration implementam a interface java.time.temporal.TemporalAmount, o que também os torna muito semelhantes.

Classe LocalDate

A chave para aprender como usar esta classe é ter em mente que ela contém o ano, mês, dia e informações derivadas de uma data. Todos os seus métodos usam essas informações ou têm uma versão para trabalhar com cada uma delas.

Os seguintes são os métodos mais importantes (mais usados) desta classe.

Para criar uma instância, podemos usar o método estático of:

```java
// Com ano (-999999999 a 999999999),
// mês (1 a 12), dia do mês (1 a 31)
LocalDate newYear2001 = LocalDate.of(2001, 1, 1);
// Esta versão usa o enum java.time.Month
LocalDate newYear2002 = LocalDate.of(2002, Month.JANUARY, 1);
```

Note que, ao contrário de java.util.Date, os meses começam em um. Se você tentar criar uma data com valores inválidos (como 29 de fevereiro), uma exceção será lançada. Para a data de hoje use now():

```java
LocalDate today = LocalDate.now();
```

Uma vez que temos uma instância de LocalDate, podemos obter o ano, o mês e o dia com métodos como os seguintes:

```java
int year = today.getYear();
int month = today.getMonthValue();
Month monthAsEnum = today.getMonth(); // como enum
int dayYear = today.getDayOfYear();
int dayMonth = today.getDayOfMonth();
DayOfWeek dayWeekEnum = today.getDayOfWeek(); // como enum
```

Também podemos usar o método get:

```java
int get(java.time.TemporalField field) // valor como int
long getLong(java.time.TemporalField field) // valor como long
```

Que recebe uma implementação da interface java.time.TemporalField para acessar um campo específico de uma data. java.time.ChronoField é uma enumeração que implementa esta interface, então podemos ter, por exemplo:

```java
int year = today.get(ChronoField.YEAR);
int month = today.get(ChronoField.MONTH_OF_YEAR);
int dayYear = today.get(ChronoField.DAY_OF_YEAR);
int dayMonth = today.get(ChronoField.DAY_OF_MONTH);
int dayWeek = today.get(ChronoField.DAY_OF_WEEK);
long dayEpoch = today.getLong(ChronoField.EPOCH_DAY);
```

Os valores suportados para ChronoField são:

```sql
DAY_OF_WEEK
ALIGNED_DAY_OF_WEEK_IN_MONTH
ALIGNED_DAY_OF_WEEK_IN_YEAR
DAY_OF_MONTH
DAY_OF_YEAR
EPOCH_DAY
ALIGNED_WEEK_OF_MONTH
ALIGNED_WEEK_OF_YEAR
MONTH_OF_YEAR
PROLEPTIC_MONTH
YEAR_OF_ERA
YEAR
ERA
```

Usar um valor diferente lançará uma exceção. O mesmo ocorre ao obter um valor que não cabe em um int com get(TemporalField).

Para verificar um LocalDate contra outra instância, temos três métodos, além de um para anos bissextos:

```java
boolean after = newYear2001.isAfter(newYear2002); // false
boolean before = newYear2001.isBefore(newYear2002); // true
boolean equal = newYear2001.equals(newYear2002); // false
boolean leapYear = newYear2001.isLeapYear(); // false
```

Uma vez que uma instância desta classe é criada, não podemos modificá-la, mas podemos criar outra instância a partir de uma existente.

Uma forma é através do método with() e suas versões:

```java
LocalDate newYear2003 = newYear2001.with(ChronoField.YEAR, 2003);
LocalDate newYear2004 = newYear2001.withYear(2004);
LocalDate december2001 = newYear2001.withMonth(12);
LocalDate february2001 = newYear2001.withDayOfYear(32);
```

```java
// Como esses métodos retornam uma nova instância, podemos encadeá-los!
LocalDate xmas2001 = newYear2001.withMonth(12).withDayOfMonth(25);
```

Outra forma é adicionando ou subtraindo — adivinhe — anos, meses, dias ou até semanas:

```java
// Adicionando
LocalDate newYear2005 = newYear2001.plusYears(4);
LocalDate march2001 = newYear2001.plusMonths(2);
LocalDate january15_2001 = newYear2001.plusDays(14);
LocalDate lastWeekJanuary2001 = newYear2001.plusWeeks(3);
LocalDate newYear2006 = newYear2001.plus(5, ChronoUnit.YEARS);

// Subtraindo
LocalDate newYear2000 = newYear2001.minusYears(1);
LocalDate nov2000 = newYear2001.minusMonths(2);
LocalDate dec30_2000 = newYear2001.minusDays(2);
LocalDate lastWeekDec2001 = newYear2001.minusWeeks(1);
LocalDate newYear1999 = newYear2001.minus(2, ChronoUnit.YEARS);
```

Note que as versões plus e minus usam a enumeração java.time.temporal.ChronoUnit, diferente de java.time.ChronoField. Os valores suportados são:

```nginx
DAYS
WEEKS
MONTHS
YEARS
DECADES
CENTURIES
MILLENNIA
ERAS
```

Por fim, o método toString() retorna a data no formato uuuu-MM-dd:

```java
System.out.println(newYear2001.toString()); // Imprime 2001-01-01
```

Classe LocalTime

A chave para aprender a usar esta classe é ter em mente que ela contém a hora, minutos, segundos e nanossegundos. Todos os seus métodos usam essas informações ou têm uma versão para trabalhar com cada uma delas.

A seguir estão os métodos mais importantes (mais usados) desta classe. Como você pode ver, são os mesmos (ou muito semelhantes) métodos de LocalDate, adaptados para funcionar com hora em vez de data.

Para criar uma instância, podemos usar o método estático of:

```java
// Com hora (0-23) e minutos (0-59)
LocalTime fiveThirty = LocalTime.of(5, 30);
// Com hora, minutos e segundos (0-59)
LocalTime noon = LocalTime.of(12, 0, 0);
// Com hora, minutos, segundos e nanossegundos (0-999.999.999)
LocalTime almostMidnight = LocalTime.of(23, 59, 59, 999999);
```

Se você tentar criar um horário com um valor inválido (como LocalTime.of(24, 0)), uma exceção será lançada. Para obter o horário atual, use now():

```java
LocalTime now = LocalTime.now();
```

Uma vez que temos uma instância de LocalTime, podemos obter a hora, os minutos e outras informações com métodos como os seguintes:

```java
int hour = now.getHour();
int minute = now.getMinute();
int second = now.getSecond();
int nanosecond = now.getNano();
```

Também podemos usar o método get():

```java
int get(java.time.TemporalField field) // valor como int
long getLong(java.time.TemporalField field) // valor como long
```

Assim como no caso de LocalDate, podemos ter, por exemplo:

```java
int hourAMPM = now.get(ChronoField.HOUR_OF_AMPM); // 0 - 11
int hourDay = now.get(ChronoField.HOUR_OF_DAY); // 0 - 23
int minuteDay = now.get(ChronoField.MINUTE_OF_DAY); // 0 - 1.439
int minuteHour = now.get(ChronoField.MINUTE_OF_HOUR); // 0 - 59
int secondDay = now.get(ChronoField.SECOND_OF_DAY); // 0 - 86.399
int secondMinute = now.get(ChronoField.SECOND_OF_MINUTE); // 0 - 59
long nanoDay = now.getLong(ChronoField.NANO_OF_DAY); // 0 - 86.399.999.999
int nanoSecond = now.get(ChronoField.NANO_OF_SECOND); // 0 - 999.999.999
```

Os valores suportados para ChronoField são:

```objectivec
NANO_OF_SECOND
NANO_OF_DAY
MICRO_OF_SECOND
MICRO_OF_DAY
MILLI_OF_SECOND
MILLI_OF_DAY
SECOND_OF_MINUTE
SECOND_OF_DAY
MINUTE_OF_HOUR
MINUTE_OF_DAY
HOUR_OF_AMPM
CLOCK_HOUR_OF_AMPM
HOUR_OF_DAY
CLOCK_HOUR_OF_DAY
AMPM_OF_DAY
```

Usar um valor diferente lançará uma exceção. O mesmo ocorre ao obter um valor que não cabe em um int com get(TemporalField).

Para comparar um objeto LocalTime com outro, temos três métodos:

```java
boolean after = fiveThirty.isAfter(noon); // false
boolean before = fiveThirty.isBefore(noon); // true
boolean equal = noon.equals(almostMidnight); // false
```

Assim como LocalDate, uma vez que uma instância de LocalTime é criada, não podemos modificá-la, mas podemos criar outra instância a partir de uma existente.

Uma forma é através do método with e suas versões:

```java
LocalTime ten = noon.with(ChronoField.HOUR_OF_DAY, 10);
LocalTime eight = noon.withHour(8);
LocalTime twelveThirty = noon.withMinute(30);
LocalTime thirtyTwoSeconds = noon.withSecond(32);
// Como esses métodos retornam uma nova instância, podemos encadeá-los!
LocalTime secondsNano = noon.withSecond(20).withNano(999999);
```

Claro, outra maneira é adicionando ou subtraindo horas, minutos, segundos ou nanossegundos:

```java
// Adicionando
LocalTime sixThirty = fiveThirty.plusHours(1);
LocalTime fiveForty = fiveThirty.plusMinutes(10);
LocalTime plusSeconds = fiveThirty.plusSeconds(14);
LocalTime plusNanos = fiveThirty.plusNanos(99999999);
LocalTime sevenThirty = fiveThirty.plus(2, ChronoUnit.HOURS);

// Subtraindo
LocalTime fourThirty = fiveThirty.minusHours(1);
LocalTime fiveTen = fiveThirty.minusMinutes(20);
LocalTime minusSeconds = fiveThirty.minusSeconds(2);
LocalTime minusNanos = fiveThirty.minusNanos(1);
LocalTime fiveTwenty = fiveThirty.minus(10, ChronoUnit.MINUTES);
```

Note que as versões plus e minus usam a enumeração java.time.temporal.ChronoUnit, diferente de java.time.ChronoField. Os valores suportados são:

```nginx
NANOS
MICROS
MILLIS
SECONDS
MINUTES
HOURS
HALF_DAYS
```

Por fim, o método toString() retorna o horário no formato HH:mm:ss.SSSSSSSSS, omitindo as partes com valor zero (por exemplo, retornando apenas HH:mm se os segundos/nanossegundos forem zero):

```java
System.out.println(fiveThirty.toString()); // Imprime 05:30
```

Classe LocalDateTime

A chave para aprender a usar esta classe é lembrar que ela combina as classes LocalDate e LocalTime.

Ela representa tanto uma data quanto uma hora, com informações como ano, mês, dia, horas, minutos, segundos e nanossegundos. Outros campos, como dia do ano, dia da semana e semana do ano também podem ser acessados.

Para criar uma instância, podemos usar o método estático of() ou a partir de uma instância de LocalDate ou LocalTime:

```java
// Definindo segundos e nanossegundos como zero
LocalDateTime dt1 = LocalDateTime.of(2014, 9, 19, 14, 5);
// Definindo nanossegundos como zero
LocalDateTime dt2 = LocalDateTime.of(2014, 9, 19, 14, 5, 20);
// Definindo todos os campos
LocalDateTime dt3 = LocalDateTime.of(2014, 9, 19, 14, 5, 20, 9);
// Supondo esta data
LocalDate date = LocalDate.now();
// E este horário
LocalTime time = LocalTime.now();
// Combine a data acima com o horário fornecido assim:
LocalDateTime dt4 = date.atTime(14, 30, 59, 999999);
// Ou assim
LocalDateTime dt5 = date.atTime(time);
// Combine este horário com a data fornecida. Note que LocalTime
// só possui este construtor para ser combinado com um LocalDate
LocalDateTime dt6 = time.atDate(date);
```

Se você tentar criar uma instância com um valor ou data inválida, uma exceção será lançada. Para obter a data/hora atual, use now():

```java
LocalDateTime now = LocalDateTime.now();
```

Uma vez que temos uma instância de LocalDateTime, podemos obter as informações com os métodos que já conhecemos de LocalDate e LocalTime, como:

```java
int year = now.getYear();
int dayYear = now.getDayOfYear();
int hour = now.getHour();
int minute = now.getMinute();
```

Também podemos usar o método get():

```java
int get(java.time.TemporalField field)
long getLong(java.time.TemporalField field)
```

Por exemplo:

```java
int month = now.get(ChronoField.MONTH_OF_YEAR);
int minuteHour = now.get(ChronoField.MINUTE_OF_HOUR);
```

Os valores suportados para ChronoField são:

```objectivec
NANO_OF_SECOND
NANO_OF_DAY
MICRO_OF_SECOND
MICRO_OF_DAY
MILLI_OF_SECOND
MILLI_OF_DAY
SECOND_OF_MINUTE
SECOND_OF_DAY
MINUTE_OF_HOUR
MINUTE_OF_DAY
HOUR_OF_AMPM
CLOCK_HOUR_OF_AMPM
HOUR_OF_DAY
CLOCK_HOUR_OF_DAY
AMPM_OF_DAY
DAY_OF_WEEK
ALIGNED_DAY_OF_WEEK_IN_MONTH
ALIGNED_DAY_OF_WEEK_IN_YEAR
DAY_OF_MONTH
DAY_OF_YEAR
EPOCH_DAY
ALIGNED_WEEK_OF_MONTH
ALIGNED_WEEK_OF_YEAR
MONTH_OF_YEAR
PROLEPTIC_MONTH
YEAR_OF_ERA
YEAR
ERA
```

Usar um valor diferente lançará uma exceção. O mesmo ocorre ao obter um valor que não cabe em um int com get(TemporalField).

Para verificar um objeto LocalDateTime em relação a outro, temos três métodos:

```java
boolean after = now.isAfter(dt1); // true
boolean before = now.isBefore(dt1); // false
boolean equal = now.equals(dt1); // false
```

Uma vez que uma instância de LocalDateTime é criada, não podemos modificá-la, mas podemos criar outra instância a partir de uma existente.

Uma forma é através do método with e suas versões:

```java
LocalDateTime dt7 = now.with(ChronoField.HOUR_OF_DAY, 10);
LocalDateTime dt8 = now.withMonth(8);
// Como esses métodos retornam uma nova instância, podemos encadeá-los!
LocalDateTime dt9 = now.withYear(2013).withMinute(0);
```

Outra maneira é adicionando ou subtraindo anos, meses, dias, semanas, horas, minutos, segundos ou nanossegundos:

```java
// Adicionando
LocalDateTime dt10 = now.plusYears(4);
LocalDateTime dt11 = now.plusWeeks(3);
LocalDateTime dt12 = now.plus(2, ChronoUnit.HOURS);

// Subtraindo
LocalDateTime dt13 = now.minusMonths(2);
LocalDateTime dt14 = now.minusNanos(1);
LocalDateTime dt15 = now.minus(10, ChronoUnit.SECONDS);
```

Nesse caso, os valores suportados para ChronoUnit são:

```nginx
NANOS
MICROS
MILLIS
SECONDS
MINUTES
HOURS
HALF_DAYS
DAYS
WEEKS
MONTHS
YEARS
DECADES
CENTURIES
MILLENNIA
ERAS
```

Por fim, o método toString() retorna a data-hora no formato uuuu-MM-dd'T'HH:mm:ss.SSSSSSSSS, omitindo as partes com valor zero. Por exemplo:

```java
System.out.println(dt1.toString()); // Imprime 2014-09-19T14:05
```

Classe Instant

Embora, em termos práticos, uma instância de LocalDateTime represente um instante na linha do tempo, há outra classe que pode ser mais apropriada.

No Java 8, a classe java.time.Instant representa um instante no número de segundos que se passaram desde a época (epoch), uma convenção usada em sistemas UNIX/POSIX e definida à meia-noite de 1º de janeiro de 1970 no horário UTC.

A partir dessa data, o tempo é medido em 86.400 segundos por dia. Essa informação é armazenada como um long. A classe também oferece suporte a precisão de nanossegundos, armazenada como um int.

Você pode criar uma instância desta classe com os seguintes métodos:

```java
// Definindo segundos
Instant fiveSecondsAfterEpoch = Instant.ofEpochSecond(5);
// Definindo segundos e nanossegundos (podem ser negativos)
Instant sixSecTwoNanBeforeEpoch = Instant.ofEpochSecond(-6, -2);
// Definindo milissegundos após (também pode ser antes) a época
Instant fifthyMilliSecondsAfterEpoch = Instant.ofEpochMilli(50);
```

Para obter a instância atual do relógio do sistema, use:

```java
Instant now = Instant.now();
```

Uma vez que temos uma instância de Instant, podemos obter as informações com os seguintes métodos:

```java
long seconds = now.getEpochSecond(); // Obtém os segundos
int nanos1 = now.getNano(); // Obtém os nanossegundos
// Obtém o valor como um int
int milis = now.get(ChronoField.MILLI_OF_SECOND);
// Obtém o valor como um long
long nanos2 = now.getLong(ChronoField.NANO_OF_SECOND);
```

Os valores de ChronoField suportados são:

```nginx
NANO_OF_SECOND
MICRO_OF_SECOND
MILLI_OF_SECOND
INSTANT_SECONDS
```

Usar qualquer outro valor lançará uma exceção. O mesmo ocorre ao obter um valor que não cabe em um int com get(TemporalField).

Para verificar um objeto Instant em relação a outro, temos três métodos:

```java
boolean after = now.isAfter(fiveSecondsAfterEpoch); // true
boolean before = now.isBefore(fiveSecondsAfterEpoch); // false
boolean equal = now.equals(fiveSecondsAfterEpoch); // false
```

Uma vez que uma instância deste objeto é criada, não podemos modificá-la, mas podemos criar outra instância a partir de uma existente.

Uma forma é através do método with:

```java
Instant i1 = now.with(ChronoField.NANO_OF_SECOND, 10);
```

Outra forma é adicionando ou subtraindo segundos, milissegundos ou nanossegundos:

```java
// Adicionando
Instant dt10 = now.plusSeconds(400);
Instant dt11 = now.plusMillis(98622200);
Instant dt12 = now.plusNanos(3000138900);
Instant dt13 = now.plus(2, ChronoUnit.MINUTES);

// Subtraindo
Instant dt14 = now.minusSeconds(2);
Instant dt15 = now.minusMillis(1);
Instant dt16 = now.minusNanos(1);
Instant dt17 = now.minus(10, ChronoUnit.SECONDS);
```

Os valores suportados de ChronoUnit são:

```nginx
NANOS
MICROS
MILLIS
SECONDS
MINUTES
HOURS
HALF_DAYS
DAYS
```

Por fim, o método toString() retorna a instância no formato uuuu-MM-dd'T'HH:mm:ss.SSSSSSSSS, por exemplo:

```java
// Imprime 1970-01-01T00:00:00.050Z
System.out.println(fifthyMilliSecondsAfterEpoch.toString());
```

Note que contém informação de zona de tempo (Z). Isso ocorre porque Instant representa um ponto no tempo a partir da época de 1970-01-01Z no fuso horário UTC.

Classe Period

A classe java.time.Period representa uma quantidade de tempo em termos de anos, meses e dias.

Você pode criar uma instância dessa classe com os seguintes métodos of:

```java
// Definindo anos, meses, dias (podem ser negativos)
Period period5y4m3d = Period.of(5, 4, 3);
// Definindo dias (podem ser negativos), anos e meses serão zero
Period period2d = Period.ofDays(2);
// Definindo meses (podem ser negativos), anos e dias serão zero
Period period2m = Period.ofMonths(2);
// Definindo semanas (podem ser negativas). O período resultante será
// em dias (1 semana = 7 dias). Anos e meses serão zero
Period period14d = Period.ofWeeks(2);
// Definindo anos (podem ser negativos), dias e meses serão zero
Period period2y = Period.ofDays(2);
```

Um Period também pode ser considerado como a diferença entre dois LocalDate. Felizmente, há um método que suporta esse conceito:

```java
LocalDate march2003 = LocalDate.of(2003, 3, 1);
LocalDate may2003 = LocalDate.of(2003, 5, 1);
Period dif = Period.between(march2003, may2003); // 2 meses
```

A data inicial é INCLUÍDA, mas a data final NÃO.

Cuidado com a forma como a data é calculada.

Primeiro, os meses completos são contados, e então o número restante de dias é calculado. O número de meses é então dividido em anos (1 ano equivale a 12 meses). Um mês é considerado se o dia final do mês for maior ou igual ao dia inicial do mês.

O resultado desse método pode ser um período negativo se a data final for anterior à data inicial (ano, mês e dia terão sinal negativo).

Aqui estão alguns exemplos:

```java
// dif1 será 1 ano 2 meses 2 dias
Period dif1 = Period.between(LocalDate.of(2000, 2, 10), LocalDate.of(2001, 4, 12));
// dif2 será 25 dias
Period dif2 = Period.between(LocalDate.of(2013, 5, 9), LocalDate.of(2013, 6, 3));
// dif3 será -2 anos -3 dias
Period dif3 = Period.between(LocalDate.of(2014, 11, 3), LocalDate.of(2012, 10, 31));
```

Uma vez que temos uma instância de Period, podemos obter as informações com os seguintes métodos:

```java
int days = period5y4m3d.getDays();
int months = period5y4m3d.getMonths();
int year = period5y4m3d.getYears();
int days2 = period5y4m3d.get(ChronoUnit.DAYS);
```

Os valores suportados de ChronoUnit são:

```nginx
DAYS
MONTHS
YEARS
```

Usar qualquer outro valor lançará uma exceção.

Uma vez que uma instância de Period é criada, não podemos modificá-la, mas podemos criar outra instância a partir de uma existente.

Uma forma é através do método with e suas versões:

```java
Period period8d = period2d.withDays(8);
// Como esses métodos retornam uma nova instância, podemos encadeá-los!
Period period2y1m2d = period2d.withYears(2).withMonths(1);
```

Outra maneira é adicionando ou subtraindo anos, meses ou dias:

```java
// Adicionando
Period period9y4m3d = period5y4m3d.plusYears(4);
Period period5y7m3d = period5y4m3d.plusMonths(3);
Period period5y4m6d = period5y4m3d.plusDays(3);
Period period7y4m3d = period5y4m3d.plus(period2y);

// Subtraindo
Period period5y4m3d = period5y4m3d.minusYears(2);
Period period5y4m3d = period5y4m3d.minusMonths(1);
Period period5y4m3d = period5y4m3d.minusDays(1);
Period period5y4m3d = period5y4m3d.minus(period2y);
```

Os métodos plus e minus aceitam uma implementação da interface java.time.temporal.TemporalAmount (ou seja, outra instância de Period ou uma instância de Duration).

Por fim, o método toString() retorna o período no formato PNYNMND, por exemplo:

```java
System.out.println(period5y4m3d.toString()); // Imprime P5Y4M3D
```

Um período zero será representado como zero dias: P0D.

Classe Duration

A classe java.time.Duration é como a classe Period, com a única diferença de que ela representa uma quantidade de tempo em termos de segundos e nanossegundos.

Você pode criar uma instância desta classe com os seguintes métodos of:

```java
Duration oneDay = Duration.ofDays(1);         // 1 dia = 86400 segundos
Duration oneHour = Duration.ofHours(1);       // 1 hora = 3600 segundos
Duration oneMin = Duration.ofMinutes(1);      // 1 minuto = 60 segundos
Duration tenSeconds = Duration.ofSeconds(10);

// Define segundos e nanossegundos (se estiverem fora do intervalo
// de 0 a 999.999.999, os segundos serão alterados, como abaixo)
Duration twoSeconds = Duration.ofSeconds(1, 1000000000);

// Segundos e nanossegundos são extraídos dos milissegundos passados
Duration oneSecondFromMilis = Duration.ofMillis(2);

// Segundos e nanossegundos são extraídos dos nanossegundos passados
Duration oneSecondFromNanos = Duration.ofNanos(1000000000);

Duration oneSecond = Duration.of(1, ChronoUnit.SECONDS);
```

Os valores válidos de ChronoUnit são:

```nginx
NANOS
MICROS
MILLIS
SECONDS
MINUTES
HOURS
HALF_DAYS
DAYS
```

Uma Duration também pode ser criada como a diferença entre duas implementações da interface java.time.temporal.Temporal, desde que suportem segundos (e, para mais precisão, nanossegundos), como LocalTime, LocalDateTime e Instant. Assim, podemos ter algo como:

```java
Duration dif = Duration.between(
    Instant.ofEpochSecond(123456789),
    Instant.ofEpochSecond(123456799)
);
```

O resultado pode ser negativo se o fim for antes do início. Uma duração negativa é expressa com sinal negativo na parte dos segundos. Por exemplo, uma duração de -100 nanossegundos é armazenada como -1 segundo mais 999.999.900 nanossegundos.

Se os objetos forem de tipos diferentes, então a duração é calculada com base no tipo do primeiro objeto. Mas isso só funciona se o primeiro argumento for um LocalTime e o segundo um LocalDateTime (porque pode ser convertido para LocalTime). Caso contrário, uma exceção é lançada.

Uma vez que temos uma instância de Duration, podemos obter as informações com os seguintes métodos:

```java
// Parte de nanossegundos da duração, de 0 a 999.999.999
int nanos = oneSecond.getNano();

// Parte de segundos da duração, positiva ou negativa
int seconds = oneSecond.getSeconds();

// Suporta SECONDS e NANOS. Outras unidades lançam exceção
int oneSec = oneSecond.get(ChronoUnit.SECONDS);
```

Uma vez que uma instância de Duration é criada, não podemos modificá-la, mas podemos criar outra instância a partir de uma existente.

Uma forma é através do método with e suas versões:

```java
Duration duration1sec8nan = oneSecond.withNanos(8);
Duration duration2sec1nan = oneSecond.withSeconds(2).withNanos(1);
```

Outra forma é adicionando ou subtraindo dias, horas, minutos, segundos, milissegundos ou nanossegundos:

```java
// Adicionando
Duration plus4Days = oneSecond.plusDays(4);
Duration plus3Hours = oneSecond.plusHours(3);
Duration plus3Minutes = oneSecond.plusMinutes(3);
Duration plus3Seconds = oneSecond.plusSeconds(3);
Duration plus3Millis = oneSecond.plusMillis(3);
Duration plus3Nanos = oneSecond.plusNanos(3);
Duration plusAnotherDuration = oneSecond.plus(twoSeconds);
Duration plusChronoUnits = oneSecond.plus(1, ChronoUnit.DAYS);

// Subtraindo
Duration minus4Days = oneSecond.minusDays(4);
Duration minus3Hours = oneSecond.minusHours(3);
Duration minus3Minutes = oneSecond.minusMinutes(3);
Duration minus3Seconds = oneSecond.minusSeconds(3);
Duration minus3Millis = oneSecond.minusMillis(3);
Duration minus3Nanos = oneSecond.minusNanos(3);
Duration minusAnotherDuration = oneSecond.minus(twoSeconds);
Duration minusChronoUnits = oneSecond.minus(1, ChronoUnit.DAYS);
```

Os métodos plus e minus aceitam outra Duration ou um valor válido de ChronoUnit (os mesmos usados para criar uma instância).

Por fim, o método toString() retorna a duração com o formato PTnHnMnS. Quaisquer segundos fracionários são colocados após um ponto decimal na seção dos segundos. Se uma seção tiver valor zero, ela é omitida. Por exemplo:

- 2 dias e 4 minutos → PT48H4M

- 45 segundos e 99 milissegundos → PT45.099S

Pontos-chave

LocalDate, LocalTime, LocalDateTime, Instant, Period, Duration são as classes principais da nova API de Data/Hora do Java, localizada no pacote java.time.
Elas são imutáveis, seguras para threads e, com exceção de Instant, não armazenam nem representam um fuso horário.

LocalDate, LocalTime, LocalDateTime e Instant implementam a interface java.time.temporal.Temporal, portanto todos têm métodos semelhantes. Enquanto Period e Duration implementam a interface java.time.temporal.TemporalAmount, o que também os torna muito semelhantes.

- LocalDate representa uma data com informações de ano, mês e dia do mês.
  Você pode criar uma instância usando:
  LocalDate.of(2015, 8, 1);

Valores válidos de ChronoField para usar com o método get() são:
DAY_OF_WEEK, ALIGNED_DAY_OF_WEEK_IN_MONTH, ALIGNED_DAY_OF_WEEK_IN_YEAR, DAY_OF_MONTH, DAY_OF_YEAR, EPOCH_DAY, ALIGNED_WEEK_OF_MONTH, ALIGNED_WEEK_OF_YEAR, MONTH_OF_YEAR, PROLEPTIC_MONTH, YEAR_OF_ERA, YEAR, e ERA.

Valores válidos de ChronoUnit para usar com os métodos plus() e minus() são:
DAYS, WEEKS, MONTHS, YEARS, DECADES, CENTURIES, MILLENNIA e ERAS.

- LocalTime representa um horário com informações de hora, minutos, segundos e nanossegundos.
  Você pode criar uma instância usando:
  LocalTime.of(14, 20, 50, 99999);

Valores válidos de ChronoField para usar com o método get() são:
NANO_OF_SECOND, NANO_OF_DAY, MICRO_OF_SECOND, MICRO_OF_DAY, MILLI_OF_SECOND, MILLI_OF_DAY, SECOND_OF_MINUTE, SECOND_OF_DAY, MINUTE_OF_HOUR, MINUTE_OF_DAY, HOUR_OF_AMPM, CLOCK_HOUR_OF_AMPM, HOUR_OF_DAY, CLOCK_HOUR_OF_DAY, e AMPM_OF_DAY.

Valores válidos de ChronoUnit para usar com os métodos plus() e minus() são:
NANOS, MICROS, MILLIS, SECONDS, MINUTES, HOURS e HALF_DAYS.

- LocalDateTime é uma combinação de LocalDate e LocalTime.
  Você pode criar uma instância usando:
  LocalDateTime.of(2015, 8, 1, 14, 20, 50, 99999);

Valores válidos de ChronoField e ChronoUnit são uma combinação dos usados para LocalDate e LocalTime.

- Instant representa um ponto único no tempo em segundos e nanossegundos.
  Você pode criar uma instância usando:
  Instant.ofEpochSecond(134556767, 999999999);

Valores válidos de ChronoField para usar com o método get() são:
NANO_OF_SECOND, MICRO_OF_SECOND, MILLI_OF_SECOND, e INSTANT_SECONDS.

Valores válidos de ChronoUnit para usar com os métodos plus() e minus() são:
NANOS, MICROS, MILLIS, SECONDS, MINUTES, HOURS, HALF_DAYS, e DAYS.

- Period representa uma quantidade de tempo em termos de anos, meses e dias.
  Você pode criar uma instância usando:
  Period.of(3, 12, 30);

Valores válidos de ChronoUnit para usar com o método get() são:
DAYS, MONTHS, YEARS.

- Duration representa uma quantidade de tempo em termos de segundos e nanossegundos.
  Você pode criar uma instância usando:
  Duration.ofSeconds(50, 999999);

Valores válidos de ChronoUnit para usar com o construtor e os métodos get(), plus() e minus() são:
NANOS, MICROS, MILLIS, SECONDS, MINUTES, HOURS, HALF_DAYS, e DAYS.

Autoavaliação

1. Quais das seguintes são formas válidas de criar um objeto LocalDate?
A. LocalDate.of(2014);
B. LocalDate.with(2014, 1, 30);
C. LocalDate.of(2014, 0, 30);
D. LocalDate.now().plusDays(5);

2. Dado:

```java
LocalDate.of(2014, 1, 2).atTime(14, 30, 59, 999999)
```

Qual dos seguintes é o resultado da execução da linha acima?
A. Um LocalDate de 2014-01-02
B. Um LocalTime de 14:30:59:999999
C. Um LocalDateTime de 2014-01-02 14:30:59:999999
D. Uma exceção é lançada

3. Quais das seguintes são valores válidos de ChronoUnit para LocalTime?
A. YEAR
B. NANOS
C. DAY
D. HALF_DAYS

4. Quais das seguintes afirmações são verdadeiras?
A. java.time.Period implementa java.time.temporal.Temporal
B. java.time.Instant implementa java.time.temporal.Temporal
C. LocalDate e LocalTime são seguros para threads
D. LocalDateTime.now() retornará a hora atual no fuso horário UTC

5. Qual das seguintes é uma forma válida de obter a parte de nanossegundos de um objeto Instant referenciado por i?
A. int nanos = i.getNano();
B. long nanos = i.get(ChronoField.NANOS);
C. long nanos = i.get(ChronoUnit.NANOS);
D. int nanos = i.getEpochNano();

6. Dado:

```java
System.out.println(
    Period.between(
        LocalDate.of(2015, 3, 20),
        LocalDate.of(2015, 2, 20))
);
```

Qual dos seguintes é o resultado da execução da linha acima?

A. P29D
B. P-29D
C. P1M
D. P-1M

7. Dado:

```java
System.out.println(
    Duration.between(
        LocalDateTime.of(2015, 3, 20, 18, 0),
        LocalTime.of(18, 5) )
);
```

Qual dos seguintes é o resultado da execução da linha acima?

A. PT5M
B. PT-5M
C. PT300S
D. Uma exceção é lançada

8. Quais das seguintes são valores válidos de ChronoField para LocalDate?

A. DAY_OF_WEEK
B. HOUR_OF_DAY
C. DAY_OF_MONTH
D. MILLI_OF_SECOND